IV.2 Syntax von Prolog

Dienstag, 16. Januar 2018 09:30

- Programm = Folge von speziellen logischen Formeln (sog. Klauseln). Nach jeder Klausel muss Zeilenumbruch (oder Leerzeichen) kommen.
- 2 Arten von Klauseln: Fakten + Regeln. Klausel besteht aus Literalen (≙ atomare Formeln).
- Literale: Prädikatssymbol angewendet auf Terme
- Prädikatssymbol hat Stelligkeit, wird oft mit Schrägstrich angegeben:

  weiblich/1
  verheiratet/2

  Symbole können überladen werden (d.h. es darf auch ein Prädikat verheiratet/1 geben, das nichts mit verheiratet/2 zu tun hat).
- Prädikatssymbole sind Strings, die mit Kleinbuchst. beginnen

- Term: Variable oder Funktionssymbol angewendet auf Terme
- Variable: String, der mit Großbuchst. oder _ anfängt.
- Anonyme Variable _ : mehrfache Vorkommen dürfen unterschiedlich belegt werden u. die Belegung von _ wird in der Antwortsubst. nicht mit ausgegeben.

```
?- verheiratet(gerd, _).
true

?- verheiratet(_,_).
true
```

- Es existieren auch vordef. Prädikatssymbole wie >, <=, ...

Prädikatssymbole: dienen zur Formulierung von Aussagen (Literale) (wahr oder falsch)

Funktionssymbole: dienen zur Konstruktion von Objekten (Terme)

z.B. monika/0, werner/0

Es darf auch Funktionssymbole mit Stelligkeit >0 geben.

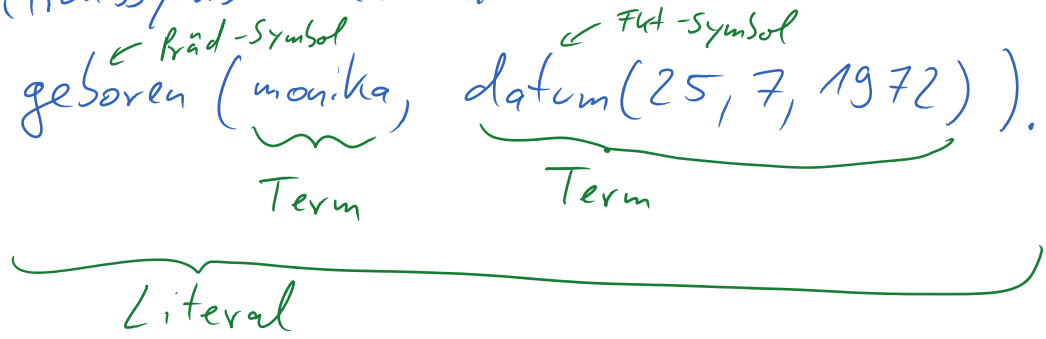
z.B.: Erweitere Programm um Aussagen zum Geburtsdatum der Personen.

1. Möglichkeit: Führe ein Prädikatssymbol geboren/4 ein:

geboren(monika, 25, 7, 1972).

Unschön. Datum sollte ein Objekt sein.

2. Möglichkeit: Führe Prädikatssymbol geboren/2 und Funktionssymbol datum/3 ein:



Syntax von Funktionssymbolen:

Strings, die mit Kleinbuchstaben beginnen

(und es ex. vordef. Funktionssymbole wie 2, +, -, ...).

Datenstrukturen in Prolog

Datenobjekte müssen durch Terme repräsentiert werden (wie in Haskell), d.h.:

Man benötigt Funktionssymbole, mit denen man alle Objekte der Datenstruktur darstellen kann.

In Haskell: Datenkonstruktoren werden nicht ausgewertet, alle anderen Funktionen werden ausgewertet.

In Prolog: Funktionen werden nicht ausgewertet, d.h. alle Funktionen entsprechen Datenkonstruktoren. Nur

Funktionen entsprechen Datenkonstruktoren. Nur Prädikate werden ausgewertet.

Bsp: Datenstruktur der nat. Zahlen

Verwende 2 Funktionssymbole zero/0 und succ/1.

Dann: zero $\hat{=}$ 0

succ(zero) $\hat{=}$ 1

succ(succ(zero)) $\hat{=}$ 2

⋮

Nun kann man Programme auf nat. Zahlen schreiben, die durch diese Terme repräsentiert werden.

Haskell: implementiere Fkt. add :: Nats -> Nats -> Nats

Prolog: Funktionen werden nicht ausgewertet.

Daher: implementiere ein Prädikat add/3:

add(X,Y,Z) $\hat{=}$ Die Addition von X und Y ergibt Z

$\hat{=}$ X+Y = Z

A: X+0 = X

B: X+(Y+1) = (Z+1), falls X+Y = Z

?-add(s(zero), s(zero), U)

| X = s(zero)
| Y = zero
| U = s(Z)

?-add(s(zero), zero, Z)

Antwort setzt ergibt sich, indem man nacheinander die Substitutionen des erfolgreichen Pfads auf die

Hier wird die

Hier wird die Prog-Klausel add(X', zero, X') benutzt.
Bei jeder Anwendung einer Prog-Klausel werden Variablen der Prog-Klausel umbenannt, so dass sie verschieden von bisher benutzten Variablen sind.

?- add(s(zero), zero, t)
| t = s(zero)
| X' = s(zero)
□

des erfolgreichen Pfads auf die Variablen in der ursprünglichen Anfrage an wendet.
D.h: ersetze U durch s(Z) und anschließend Z durch s(zero)
⇒ U = s(s(zero))

Prolog ist bidirektional: Man muss nicht 1. + 2. Argument vorgeben, sondern man könnte auch das 1. + 3. Arg. vorgeben. Man kann das add-Prog. daher auch zur Subtraktion benutzen. Genauso kann man auch Argumente "teilweise" instantiieren (durch Terme mit Variablen).

Bsp: mult/3 mult(X, Y, Z) ≙ Die Multiplik. von X und Y ergibt Z
≙ X · Y = Z

A: X · 0 = 0

B: X · (Y+1) = Z, falls X · Y = U und
X · Y + X X + U = Z
U

?- mult(s(zero), s(zero), U).

|

?- mult(s(zero), zero, U'), add(s(zero), U', U).

| U' = zero

?- add(s(zero), zero, U)

| U = s(zero)

□

Prolog arbeitet wie folgt:

- Prog-Klauseln werden von oben nach unten abgearbeitet.
- Folgen von Literalen werden von links nach rechts abgearbeitet.

D.h: Es spielt eine Rolle

- in welcher Reihenfolge die Klauseln im Prog. stehen
- — u — die Literale im Rumpf einer Regel stehen.

Bsp: Vertausche die Literale in der rekursiven mult-Regel:

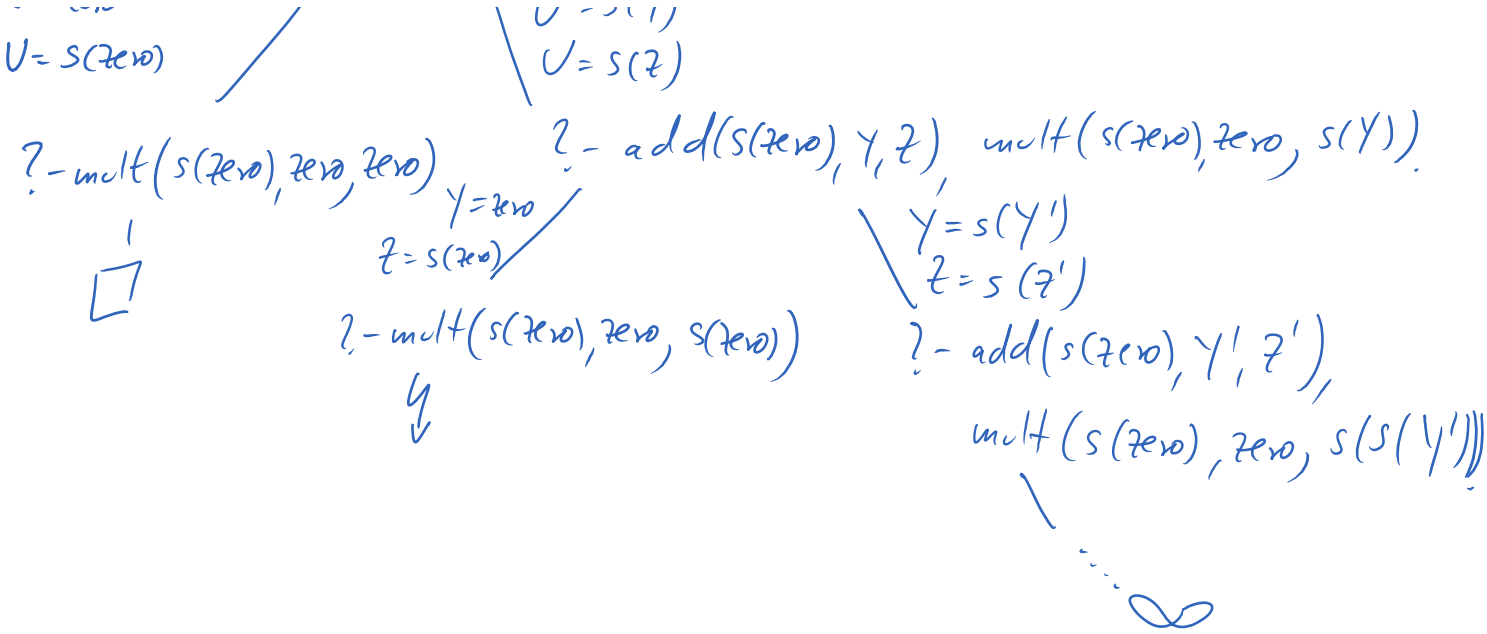
mult(X, s(Y), Z) :- add(X, U, Z), mult(X, Y, U).

?- mult(s(zero), s(zero), U).

|

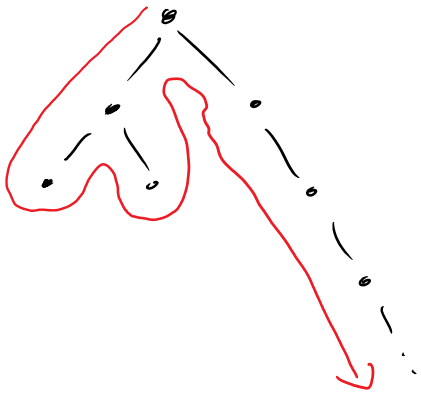
?- add(s(zero), U', U), mult(s(zero), zero, U')

U' = zero, U = s(zero) / \ U' = s(Y), U = s(Z)



⇒ Prolog findet die erste Antwort U = s(zero), aber wenn man danach ';' eingibt, dann baut Prolog den gesamten (unendlichen) Beweisbaum auf und terminiert nicht.

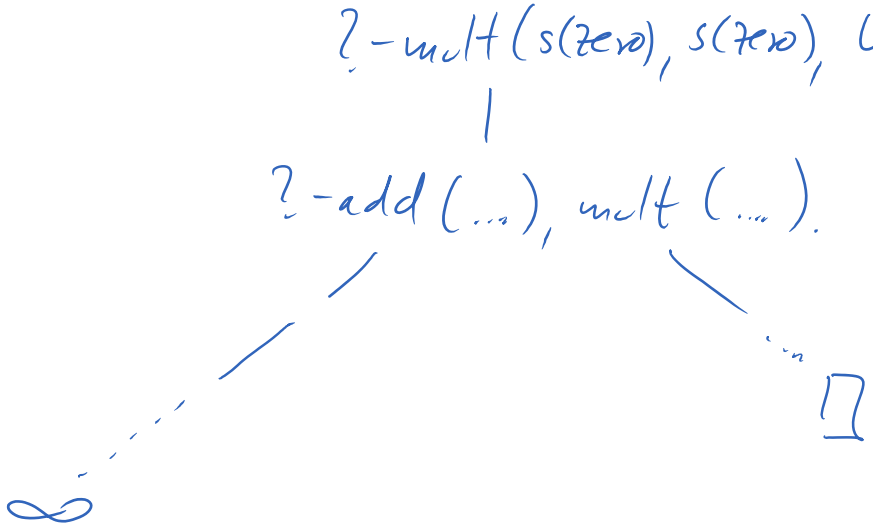
Beweisbaum wird von Prolog in Tiefensuche von links nach rechts aufgebaut:



Wenn links von einer Lösung ein unendlicher Pfad ist, dann terminiert Prolog nicht und findet die Lösung daher nicht.

Bsp: Vertausche zusätzlich die add-Klausel.

⇒



⇒ Prolog terminiert nicht und findet die Lösung nicht.

Generelle Heuristik:

- Nicht-rekursive Klauseln sollten vor rekursiven Klauseln des gleichen Prädikats.
- Ordne Literale in Rümpfen von Regeln so, dass möglichst viele ihrer Argumente belegt sind, wenn das Literal ausgewertet wird.

Prolog ist eine untypisierte Sprache ← Es gibt auch typisierte logische Programmiersprachen

```
?- mult(monika, zero, zero).
true
?- mult(succ(monika), succ(zero), succ(monika)).
true
```

Datenstruktur der Listen

Listen

nil/0 ≙ leere Liste

cons/2 ≙ Einfügen in Listen

cons(zero, cons(succ(zero), nil))

≙ [0, 1]

Programmiere "Listenlänge" nicht als 1-stellige Funktion, sondern als 2-stelliges Prädikat:

len(L, N) ≙ "Die Liste L hat die Länge N"

Prolog versucht stets, die allgemeinsten Lösungen zu finden.

?- len(L, s(s(zero))).

L = cons(A, A')
A' = s(zero)

Prog-Klausel:
len(cons(A, A'), succ(A'')) :-
len(A', A'').

?- len(A', s(zero)).

A' = cons(B, B')

Prog-Klausel:
len(cons(B, B'),

$A' = cons(B, B')$
$B'' = zero$

Prog-Klausel:

len(cons(B, B'), succ(B'')) :-
len(B', B'').

?- len(B', zero)

$B' = nil$ |

□

Antwortsubstitution:

$\sigma_1 = \{ L = cons(A, A'), \ldots \}$

$\sigma_2 = \{ A' = cons(B, B'), \ldots \}$

$\sigma_3 = \{ B' = nil \}$

Antwortsubstitution ist

$\sigma_3 \circ \sigma_2 \circ \sigma_1$

Komposition der 3 Substitutionen:
erst σ_1, dann σ_2, dann σ_3

Antw.-Subst wird nur auf die Variablen der ursprüngl. Anfrage angewendet (d.h. auf L):

$\sigma = \{ L = cons(A, cons(B, nil)) \}$

Die Variablen A und B werden automatisch erzeugt:

Bei jeder Anwendung einer Programmklausel werden

einer Programmklausel werden die Variablen darin durch frische neue Variablen ersetzt (könnte auch _157 heißen etc.)

Da Listen so häufig benutzt werden, sind sie in Prolog vordefiniert. Anstelle einer eigenen Datenstr. für Listen kann man die vordef. Datenstr. verwenden: Vorteil ist bessere Lesbarkeit, da es dafür Kurznotationen gibt.

Statt nil verwendet man [].

Statt cons — " — .

Statt

cons(zero, cons(succ(zero), nil))

schreibt man also

.(zero, .(succ(zero), [])).

Hierfür existiert die Kurzschreibweise:

[zero, succ(zero)]

Man kann auch Folgendes schreiben:

[t | l] steht für die Liste, die entsteht, wenn man t vorne in die Liste l einfügt.

[$t_1, \ldots, t_n$ | l] steht für die Liste, die aus l entsteht, wenn man vorne $t_1, \ldots, t_n$ einfügt.

[1,2,3] =

[1 | [2,3]] =

[1, 2 | [3]] =

[1, 2, 3 | [ ]] =

.(1, .(2, .(3, [ ])))